IME-USP

UNIVERSIDADE DE SÃO PAULO

INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Disciplina MAC0110 - Introdução à Computação

Professor: Roberto Hirata Junior

Análise dos resultados obtidos dos experimentos com diversos algoritmos de ordenação

Nome: Thalia Angelo Gomes da Silva

NUSP: 15489751

Turma: 45

Junho, 2024

SUMÁRIO

1 - INTRODUÇÃO

O objetivo do trabalho foi analisar como alguns algoritmos de ordenação se comportam com distintos tamanhos de elementos. Foram utilizados os algoritmos InsertionSort, SelectionSort, BubbleSort, CountingSort e o algoritmo nativo do Python TimSort.

O primeiro experimento realizado foi observar o que acontece com cada algoritmo ao passo que a lista de elementos aumenta. Ou seja, cada um dos algoritmos ordenou listas com 1000, 5000, 10000, 50000 e 100000 elementos.

O segundo experimento realizado foi observar o comportamento dos algoritmos InsertionSort, BubbleSort e TimSort quando são dadas listas com 1%, 3%, 5%, 10% e 50% de desordenação.

No terceiro experimento realizado, foi implementado os mesmos algoritmos que implementamos na linguagem Python para a linguagem C, então, foram observadas diferenças na implementação dos mesmos algoritmos em linguagens diferentes. Porém, as funções de ordenação foram chamadas na linguagem C, mas dentro da main do código em Python, ou seja, foram usadas as duas linguagens ao mesmo tempo.

Com o resultado obtido dos experimentos, será possível perceber se, para cada tamanho da lista, um dado algoritmo é mais eficiente para a ordenação e, se uma porcentagem da lista já estiver ordenada, alguns algoritmos serão mais eficientes do que outros para ordenar o restante da lista.

2 – MATERIAIS E MÉTODOS

Os algoritmos de ordenação utilizados, à primeira vista, aparentam mostrar enorme semelhança, porém, na prática, cada um ordena listas de uma maneira diferente, alguns de maneira mais eficiente e outros de maneira menos eficiente, variando de acordo com a lista dada.

O algoritmo SelectionSort compara o primeiro elemento da lista, ou seja, o valor de $V[0]$ com todos os elementos seguintes, ou seja, compara com $V[1]$, $V[2]$, ..., $V[n-1]$, e, a cada menor valor que encontra, o valor é inserido em $V[0]$, já que, no momento, é o começo da ordenação. Assim, após terminar a primeira verificação com os $(n-1)$ elementos, teremos certeza de que o menor elemento da lista estará na posição 0 da lista, e assim segue a ordenação do restante da lista. Como serão necessárias $(n-1)$comparações para o menor elemento ocupar a posição 0, $(n-2)$ comparações para o segundo menor elemento ocupar a posição 1, e assim por diante, temos que:

Total de comparações = $\sum_{i=1}^{(n-1)} i = \frac{n(n-1)}{2}$, ou seja, temos um algoritmo com complexidade $O(n^2)$.

O algoritmo BubbleSort compara dois a dois elementos, e, a cada vez que compara, o maior elemento aumenta o seu índice, ou seja, se $V[i] > V[i+1]$, o elemento de $V[i]$ passa a ser o elemento de $V[i+1]$. Ao final da primeira comparação com os $(n-1)$ elementos, teremos certeza de que o maior elemento da lista estará na posição final, ou seja, será o elemento $V[n-1]$. Novamente, como serão $(n-1)$ comparações para o maior elemento da lista ocupar a posição $(n-1)$, $(n-2)$ comparações para o segundo maior elemento ocupar a posição $(n-2)$, e assim por diante, temos que:

Total de comparações = $\sum_{i=1}^{(n-1)} i = \frac{n(n-1)}{2}$, ou seja, também teremos um algoritmo com a complexidade $O(n^2)$.

O algoritmo InsertionSort também compara dois a dois elementos. Porém, neste caso, a cada troca de elementos que acontece, é comparado do índice atual até o índice 0. Por exemplo, se $V[i] > V[i+1]$, os elementos desses índices são trocados, e então são realizadas as comparações $V[i+1] > V[i]$, ..., $V[0] > V[1]$. Ao final dessa troca, quando é realizada a comparação do índice atual até o índice 0, teremos certeza de que até o

índice i a lista estará ordenada. Como será necessária 1 comparação para $V[0]$ e $V[1]$, 2 comparações para $V[0]$, $V[1]$ e $V[2]$, ..., $(n - 1)$ comparações para $V[0]$, $V[1]$, ..., e para $V[n - 1]$, então, novamente, temos que:

Total de comparações = $\sum_{i=1}^{(n-1)} i = \frac{n(n-1)}{2}$, resultando também em um algoritmo com a complexidade $O(n^2)$.

O algoritmo CountingSort ordena uma forma mais eficiente do que os algoritmos vistos acima. Primeiro, encontramos o maior elemento de uma dada lista, após isso, uma nova lista é criada com o índice até o maior elemento acrescentado de 1, e então, a cada elemento encontrado na lista dada, acrescentamos +1 no seu respectivo índice. Ou seja, se $V[i] = n$, então, na nova lista criada, ocorre a mudança $W[n] = W[n] + 1$, assim, será incrementado a quantidade de ocorrências desse índice na lista V . E então, após a verificação de todos os elementos da lista V , uma nova lista será criada, em que adicionaremos a quantidade de vezes que um índice aparece, a partir do 0. Assim, se $W[0] = 2$, então, a nova lista será $T[0] = 0$, $T[1] = 0$, e assim segue até o índice do maior elemento acrescentado de 1 encontrado na lista V . Ao final desse procedimento, teremos certeza de que colocamos todos os índices, de forma crescente e com suas respectivas quantidades de ocorrências na nova lista. No CountingSort, não teremos comparações diretas como os algoritmos mencionados anteriormente, apenas percorreremos um loop de cada vez, então:

Total de comparações = 0, resultando em um algoritmo de complexidade $O(n)$.

O algoritmo nativo do Python, TimSort, é uma função de ordenação padrão da linguagem, em que sua complexidade é $O(n \log_2 n)$.

Para a implementação dos algoritmos descritos foi utilizado a versão 3.12 do Python, pela máquina Samsung Book, com 4 GB de memória RAM e processador Intel(R) Celeron(R) 6305 @ 1.80GHz 1.80 GHz.

3 – PERGUNTAS CIENTÍFICAS

3.1 - EXPERIMENTO 1

O primeiro experimento, como dito na introdução, foi realizado a análise do comportamento de cada algoritmo dado tamanhos diferentes de listas de elementos para serem ordenados. Então, primeiramente foi realizada a ordenação de listas com 1000, 5000, 10000, 50000 e 100000 elementos com o algoritmo de ordenação TimSort.

Em seguida, foi realizado o ordenamento com listas de mesmo tamanho com os algoritmos BubbleSort, InsertionSort, SelectionSort e CountingSort. (Note que, para cada tamanho diferente, foi utilizada a mesma lista nos algoritmos).

Através do gráfico e da tabela disponibilizadas do primeiro experimento, é possível notar que, para dados tamanhos de listas, um algoritmo leva vantagem em ser usado. Podemos ver, no gráfico, que dentre os algoritmos com complexidade de $O(n^2)$, o BubbleSort se torna menos eficiente para listas com número muito grande de elementos. Conseguintemente, o SelectionSort e o InsertionSort também diminuem muito as suas respectivas eficiências para listas com números muito grande de elementos. E para o CountingSort, é esperado que ele seja mais eficiente que os algoritmos mencionados anteriormente, mesmo que o tamanho da lista de elementos aumente, já que, dados que o BubbleSort, o InsertionSort e o SelectionSort possuem a complexidade de $O(n^2)$, o que o algoritmo CountingSort leva vantagem, por apresentar a complexidade de $O(n)$.

É possível notar que, no gráfico, os algoritmos InsertionSort e SelectionSort, apesar de ambos apresentarem a complexidade de $O(n^2)$, para este experimento, o InsertionSort se comportou de maneira semelhante ao TimSort e ao CountingSort, apesar de ambos apresentarem a complexidade de $O(n)$, o que é algo inesperado, já que, esperaríamos que o InsertionSort e o SelectionSort se comportassem de maneiras semelhantes.

Através da tabela com os resultados numéricos, a variância do BubbleSort aumenta bastante conforme é maior o número de elementos. Porém, para listas com poucas quantidades de elementos, ou seja, de acordo com o gráfico e a tabela, para listas com o número de elementos menores que 10000, os 5 algoritmos mantêm suas eficiências para ordenar as listas. Porém, quando o número de elementos tende a aumentar, o BubbleSort, SelectionSort e o InsertionSort, respectivamente, vão se tornando menos eficientes.

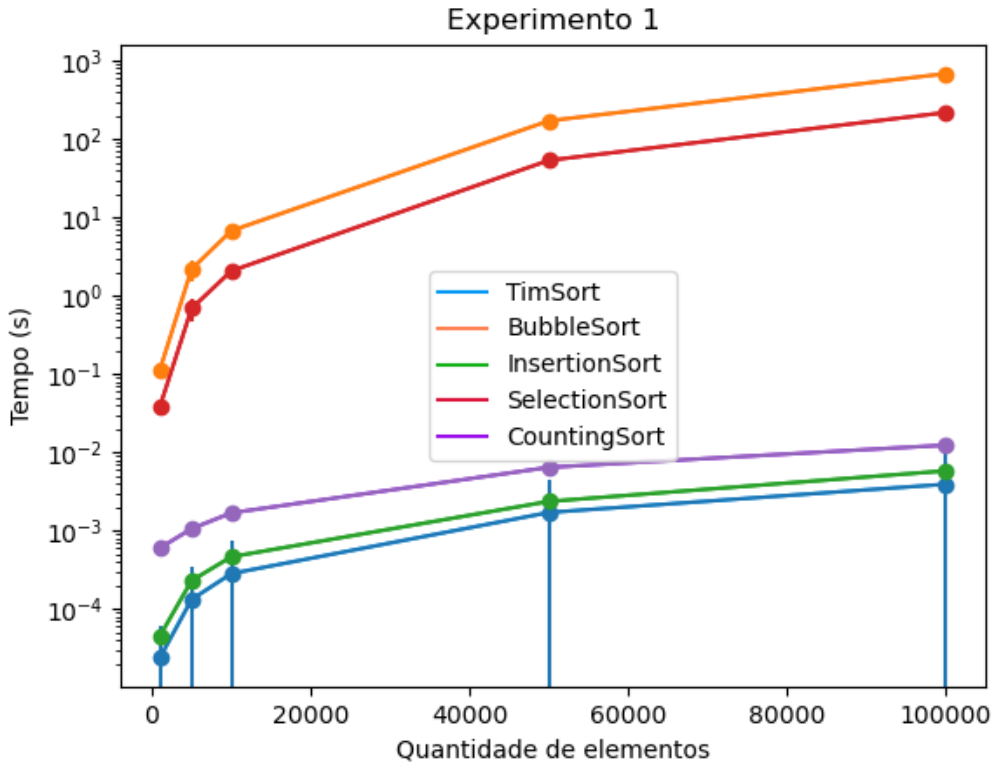

Figura 1. Gráfico do experimento 1 gerado pela função errorbar.

	1000	*5000*	*10000*	*50000*	*100000*
TimSort	0.00	0.00	0.00	0.00	0.00
BubbleSort	0.11	2.19	6.73	170.58	682.30
InsertionSort	0.10	0.22	0.41	0.89	0.97
SelectionSort	0.04	0.69	2.07	53.39	217.65
CountingSort	0.00	0.00	0.00	0.01	0.01

Figura 2. Tabela com as funções e suas respectivas médias de tempos com listas de tamanhos diferentes.

3.2 - EXPERIMENTO 2

Agora, para o segundo experimento, como também dito na introdução, foi realizada a análise do comportamento dos algoritmos TimSort, BubbleSort e InsertionSort para listas de tamanhos fixo de 100000 elementos, porém, com quantidades diferentes de desordenação.

Foi realizado, a ordenação de listas com 1% , 3%, 5%, 10% e 50% de desordenação, através dos algoritmos TimSort, BubbleSort e InsertionSort, respectivamente. (Note que foram usadas as mesmas listas para cada porcentagem diferente para os três algoritmos).

Através do gráfico e da tabela do experimento 2, é possível notar que, para o algoritmo BubbleSort, mesmo que a lista esteja 90% ordenada, conforme visto no experimento anterior, a sua eficiência aumenta conforme aumenta o número de elementos da lista. Por conta de apresentar tamanho fixo de 100000 elementos, sua complexidade se mantém $O(n^2)$, o que o torna extremamente ineficiente, conforme foi observado.

O mesmo pode ser observado para o caso do InsertionSort, novamente, como foi observado no experimento anterior, mesmo que ambos apresentem a complexidade de O(n2), o InsertionSort demonstra mais eficiência para listas com número de elementos maiores do que o BubbleSort. O TimSort, por sua vez, por apresentar a complexidade de O(n), demonstra enorme eficiência para listas com números grandes de elementos e com uma porcentagem de desordenação qualquer, já que não influencia a sua eficiência.

O esperado, para os algoritmos BubbleSort e InsertionSort era de que, pelo menos em quantidades que as listas estivessem mais ordenadas, que seriam mais eficientes para ordenar o restante. Porém, após a análise do gráfico e da tabela, que, para esses algoritmos, não é de extrema relevância se a lista já se encontra parcialmente ordenada, já que a diferença de tempos conforme aumenta a porcentagem de desordenação não varia muito.

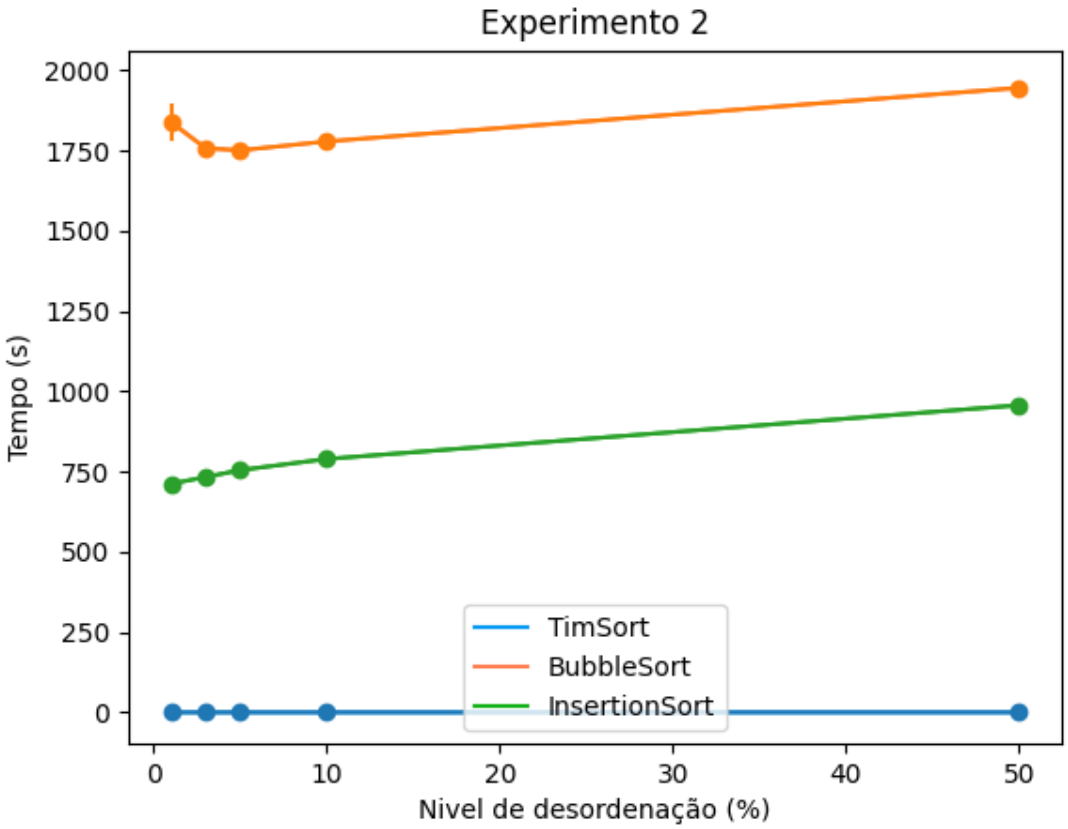

Figura 3. Gráfico do experimento 2 gerado pela função errorbar.

	1%	*3%*	*5%*	*10%*	*50%*
TimSort	0.00	0.00	0.00	0.00	0.00
BubbleSort	1827.49	1753.41	1750.51	1789.08	1962.27
InsertionSort	709.87	729.01	756.13	778.13	832.09

Figura 4. Tabela com as funções e suas respectivas médias de tempos dada uma porcentagem de desordenação.

3.3 - EXPERIMENTOS EM C

Como também dito na introdução, implementamos os algoritmos de ordenação em linguagem C e transformamos isso em uma biblioteca. Assim, conseguimos rodar as funções da linguagem C no arquivo em Python.

Ou seja, escrevemos as funções em C, e tratamos isso como uma biblioteca, assim, importamos essa biblioteca para o programa em Python, assim, conseguimos chamar as funções de ordenação no Python.

Este experimento mostrou que a linguagem C é extremamente eficiente ao se tratar de loops, em que, comparado ao experimento 1 realizado em Python visto anteriormente, o tempo é extremamente menor para as mesmas funções e listas com os mesmos tamanhos.

Agora, analisando o que ocorreu no experimento em C, como um todo, mantivemos os mesmos resultados que já esperávamos sobre a eficiência de cada algoritmo com respectivos tamanhos. Ou seja, apesar de mudar a linguagem, os algoritmos se comportam da mesma maneira, em que o CountingSort apresenta melhor eficiência conforme aumenta o tamanho das listas.

Adiante é possível analisar os resultados obtidos das ordenações, com seus respectivos tempos e tamanhos de listas.

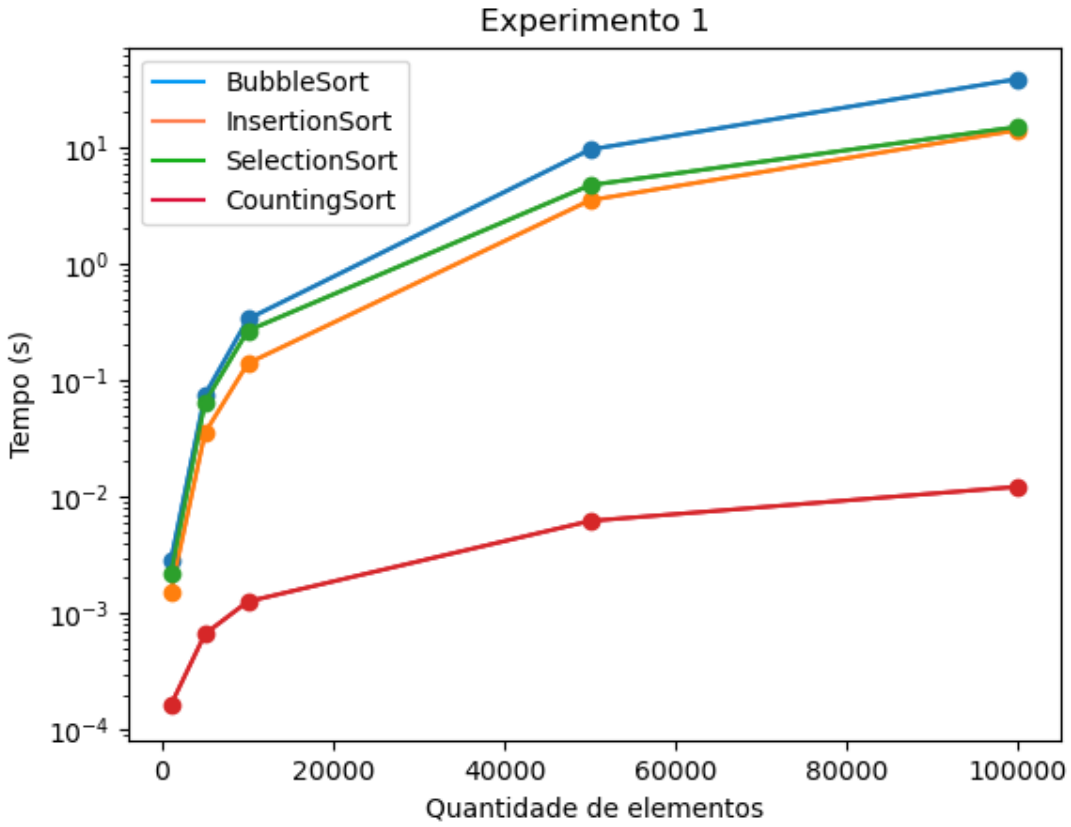

Figura 5. Gráfico do experimento com as funções na linguagem C gerados pela função errorbar.

	1000	*5000*	*10000*	*50000*	*100000*
SelectionSort	0.00	0.07	0.26	4.69	14.86
BubbleSort	0.00	0.07	0.34	9.5	38.29
InsertionSort	0.00	0.04	0.14	3.49	13.98
CountingSort	0.00	0.00	0.00	0.01	0.01

Figura 6. Tabela com as funções em C e suas respectivas médias de tempo de ordenação com listas de tamanhos variados.

4 - CONCLUSÃO

Conforme foi observado em **3.1** e em **3.2,** é notável que a eficiência de cada algoritmo varia conforme aumenta ou diminui a quantidade de elementos da lista utilizada. Também, chegamos à conclusão de que o ordenamento atual de uma lista não apresenta impacto notável para 100% de sua ordenação.

A experiência que obtive deste projeto foi novidade, já que, foi a primeira vez que tive que fazer um projeto científico nesse nível, e imagino que é algo que farei muitas vezes de agora em diante, o que estou animada para realizar.

Com este projeto também, aprendi a enorme variedade que é possível chegar através das linguagens de programação, especial o Python, utilizado para esse projeto, já que, foi a primeira vez que o utilizei para a criação de gráficos, por exemplo.

Mas, quando estava implementando as funções necessárias para o projeto, me encontrei diante de algumas dificuldades, em especial para a implementação da função timeMe(), já que, muitas coisas eu não estava conseguindo generalizar para funcionar em todos os casos, por exemplo, para receber uma função como parâmetro de uma outra função. Também tive dificuldades para a organização do main, em que, por vezes não sabia como deixar de maneira eficiente o modo que estava recebendo os valores das médias e das variâncias. Outrossim, a máquina em que rodei os experimentos, por não ser uma das melhores, piorou a eficiência do projeto, em que aumentou as horas necessárias para terminá-lo. Outra dificuldade que enfrentei também foi o fato que o primeiro algoritmo do InsertionSort em que eu criei, ele estava completamente ineficiente, então, após algumas mudanças, ele se tornou eficiente.

Outra dificuldade foi encontrada ao realizar o experimento em C, em que, na minha máquina, ao tentar rodar o programa, obtive muitos problemas, como erros do Windows ao tentar rodar o programa. Assim, foi necessário rodar este experimento nos computadores da REDE LINUX do IME.

REFERÊNCIAS

https://en.wikipedia.org/wiki/Timsort